The Application of Supervisory Control to Deadlock Avoidance in Concurrent Software

Yin Wang, Terence Kelly, Manjunath Kudlur, Scott Mahlke, and Stéphane Lafortune

Abstract— **Ensuring deadlock-free execution of concurrent programs is a notoriously difficult problem, but an increasingly important one as multicore processors compel performance-conscious software developers to parallelize applications. We propose and validate a novel methodology for dynamically controlling the execution of concurrent software in order to provably avoid deadlocks. The methodology is based on supervisory control of discrete event systems modeled by Petri nets. Specifically, we synthesize feedback controllers for concurrent programs based on the theory of supervision based on place invariants and implement the controllers online to guarantee deadlock avoidance. We describe a full implementation of this methodology and report initial experimental results demonstrating its effectiveness and scalability.**

I. INTRODUCTION

Research on building robust, reliable, secure, and highly available software has been very active in the software and operating systems research community. More than half of the papers in recent conferences such as SOSP [30] and OSDI [25] are directly or indirectly related to software defects, faults, and reliability. However, existing solutions to these problems are typically *ad hoc*, based on best-practice heuristics. Model-based solutions are rare. For example, a recent award-winning paper employs runtime control to avoid software failures by rolling back a program to a recent checkpoint and re-executing the program in a modified environment [27]. Without a program model and proper feedback, the re-execution simply tries environment modifications suggested by the nature of the failure that was detected. Another paper tries to block malicious input to prevent vulnerabilities in software programs from being exploited [2]. The input filter is generated and refined using heuristics learned from practice, without models or formal methods.

We believe that supervisory control methods developed in the field of discrete event systems offer considerable promise in many computer systems problems, provided suitable models can be built and scalability issues can be addressed. Recently, we applied supervisory control based on automata models to control workflow execution control for Information Technology automation [31]. *Classical control* techniques based on linear system models have been successfully applied to several performance-related computer problems [10]. One popular application is to stabilize the response time or throughput of server applications [26]. However, discrete event system methods are better suited to problems surrounding qualitative functional requirements.

This paper demonstrates the application of supervisory control to deadlock avoidance in concurrent software. Deadlock refers to the situation where two or more processes or threads hold a set of resources and wait for resources in the same set. Ensuring deadlock-free execution of concurrent programs is a notoriously difficult problem, but an increasingly important one as multicore processors compel performance-conscious developers to parallelize software.

Deadlock is a well-known problem in concurrent programs. Early research studied the characteristics of deadlocks and identified three classes of solutions: deadlock prevention, avoidance, and detection/recovery [4], [22]. Due to the ever-increasing complexity of software and the lack of formal models, it is difficult to apply these deadlock resolution methods. For example, it is well known that deadlocks involving locks cannot occur if threads acquire locks in a consistent order. In practice, however, it is remarkably difficult to enforce global lock ordering across the many independently-developed software modules that constitute a complex modern application. The Banker's Algorithm [3], [9] is another example revealing the gap between theory and practice. Given the maximum resources needed by each thread, the algorithm allocates resources based on safety tests. In practice, programs rarely declare all resources needed in advance, which may depend on run-time inputs, complex branching, and the execution environment.

Today's operating systems have no deadlock resolution mechanism, and lay the burden on programmers for correct program behavior. Deadlocks are hard to reason about, highly environment dependent and therefore difficult to discover, reproduce, and debug. Deadlock bugs can survive software testing undetected, and are widespread in production code, including operating system kernels [5]. The rise of multicore processors exacerbates the problem because deadlocks that usually remain latent on uniprocessors are more likely to manifest under real concurrency.

This paper considers deadlocks involving mutual exclusion locks in concurrent programs. Mutual exclusion locks (or *mutexes*) are used to enforce orderly access to shared state, e.g., global variables, by concurrent threads. When multiple threads do not correctly coordinate their acquisition of locks,

The research of YW and SL is supported in part by NSF grants CCR-0325571 and ECCS-0624821. The research of MK and SM is supported in part by NSF grant CNS-0615261. Support from HP Labs is also gratefully acknowledged.

T. Kelly is with Hewlett-Packard Labs, Palo Alto, CA 94304, USA, terence.p.kelly@hp.com

The other authors are with the Department of Electrical Engineering and Computer Science, The University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109–2122, USA, {yinw,kvman,mahlke,stephane}@eecs.umich.edu

deadlocks can occur. Mutexes are widely used in programming paradigms that support multi-threaded concurrency, both for application-level programs and for infrastructure-level programs such as operating systems. Recent research on mutex deadlocks includes the following approaches:

Static analysis checks software source code for defects. It has been applied to deadlock analysis [5] and other problems [17]. While programs do not suffer runtime overhead with static analysis methods, lack of runtime information limits the accuracy of the analysis. False positives are common.

Runtime monitoring methods complement static analysis with runtime information from sample executions. False positives do not occur, but false negatives are possible because runtime monitoring detects only deadlocks that are triggered by specific inputs and execution conditions, and exhaustive testing is infeasible. Both static analysis and runtime monitoring merely detect deadlocks. Fixing them remains manual, costly, time-consuming and error-prone.

Runtime control research dates back to the Banker's Algorithm [3], [9]. Extensions to the original algorithm have been developed [19], [32]. In general, these algorithms are not practical for two reasons: over-simplified models and expensive online computations. Recently, a checkpoint and rollback method gained popularity for regulating program behavior online [23], [27]. Although developed in a different context, this method could be applied for deadlock recovery. However, rollback is not always possible, e.g., in cases where outputs or other irrevocable events have occurred.

Transactional Memory was originally proposed in the mid-80s and has enjoyed a recent resurgence due to the popularity of multicore processors [13]. It is an alternative to locking that allows the programmer to specify blocks of code to be executed atomically and in isolation. The method is not yet mature and is not compatible with legacy code that employs locks; furthermore, is not clear that I/O within atomic sections can be supported efficiently.

This paper describes and validates a novel methodology for dynamically controlling the execution of concurrent software in order to provably avoid deadlocks. Our methodology is different from the work reviewed above, as it is model-based and it employs supervisory control methods for discrete event systems modeled by Petri nets. Specifically, feedback controllers for concurrent programs are synthesized from a detailed model of the program based on the theory of Supervision Based on Place Invariants (SBPI) and implemented online to guarantee deadlock avoidance. Petri nets were chosen as the modeling formalism for this problem, instead of automata models (which we used in [31]), to address the problem of scalability of controller synthesis for large concurrent programs by exploiting the system structure captured in the Petri net model. The same motivations have led to the use of Petri nets for deadlock avoidance in manufacturing applications; see, e.g., [28].

The specific contributions of this paper include: 1) to the best of our knowledge, the first attempt at a comprehensive implementation of supervisory control techniques in general-purpose concurrent software; 2) a new approach for modeling concurrent software in a systematic manner based on compiler techniques; 3) a novel adaptation of the SBPI method to the deadlock avoidance problem in concurrent software; 4) a full implementation of the complete methodology; and 5) an evaluation of the methodology against sample programs.

Section II presents necessary background about concurrent software and control of Petri nets. An overview of our approach, together with its associated challenges, follows in Section III. Sections IV and V present implementation details and experimental results, respectively.

II. BACKGROUND

```
0 : static void * philosopher(void *id) {
1 :   ...
2 :   if (FIRST == (int *)id) {
3 :     /* grab A first */
4 :     pthread_mutex_lock(&forkA);
5 :     pthread_mutex_lock(&forkB);
6 :   }
7 :   else {   /* grab B first */
8 :     pthread_mutex_lock(&forkB);
9 :     pthread_mutex_lock(&forkA);
10:   }
11:   eat();
12:   pthread_mutex_unlock(&forkA);
13:   pthread_mutex_unlock(&forkB);
14:   ...
15: }
16: ...
17: int main(int argc, char *argv[]) {
18:   ...
19:   pthread_create(&p1, NULL,
          philosopher, (void *)id[0]);
20:   pthread_create(&p2, NULL,
          philosopher, (void *)id[1]);
21:   ...
22: }
```

Fig. 1. Dining philosophers program with two philosophers

Mutexes are widely used for data protection in popular programming languages such as C/C++ and Java. A mutex is held by at most one thread at a time, so concurrent access to shared data can be prevented. As with physical resource deadlocks, a set of threads holding a set of locks while waiting for locks in the same set is a deadlock. Figure 1 shows a part of the C program using POSIX pthread functions to simulate the familiar deadlock-prone dining philosophers problem with two philosophers and two forks.

To avoid deadlocks using supervisory control theory, we must construct a formal model from the program. The *Control Flow Graph* (CFG) [1] generated by compilers is a good starting point for this purpose. A CFG is a graph that represents all paths that might be traversed through a program during its execution. It is generated by compilers as a byproduct when compiling a program into an executable. Each node in the graph represents a basic block, i.e., a straight-line piece of code without any jumps or jump targets;

jump targets start a basic block, and jumps end a basic block. Directed edges represent jumps in the control flow.

The Petri net modeling formalism for discrete event systems offers the best combination of mathematical formality, compactness of representation, analytical power, and scalability for the objectives of this paper. We consider only ordinary Petri nets where the weight of every arc is one. The principal supervisory control technique that we employ is the method of *Supervisory control Based on Place Invariants* (SBPI). There is a large body of results on SBPI; see, e.g., [11], [12], [20]. In order to use SBPI for deadlock avoidance, we must first discover potential deadlocks in the net based on structural features called *siphons*. A siphon is a set of places that never regains a token once it empties. A *minimal siphon* is a siphon that does not contain any other siphon. It is well known that a deadlocked Petri net contains at least one empty minimal siphon. SBPI uses linear-algebraic techniques to add new *control* places to the Petri net and connect them in such a way that the net cannot deadlock.

Newly-added places to a Petri net may introduce new siphons because they change the structure of the net. Stated differently, blocking lock acquisition requests could introduce new deadlocks. A proper feedback control algorithm must avoid both natural deadlocks in the program and deadlocks introduced by control. Multiple iterations of SBPI may therefore be necessary until no new siphon is discovered. Whether this iterative procedure converges or not depends on the Petri net structure.

Note that while a siphon is a structural property, the initial condition of the net determines whether a state with an empty siphon can be reached or not. Under a given initial condition, a siphon that is never empty in the set of reachable states is called a *controlled siphon*, otherwise it is *uncontrolled*. An important objective in control synthesis is therefore to develop techniques that identify controlled siphons and make the SBPI procedure converge quickly.

SBPI guarantees the highly desirable property of *maximal permissiveness* [12], [20]: the controller never blocks a transition unless it may lead to an undesirable state unavoidably. In our context, maximal permissiveness ideally means that the controller never postpones a lock acquisition except when necessary to ensure that deadlock cannot occur. In practice, our ability to attain this ideal is limited by the accuracy of our models. Our models of programs may be imperfect if, e.g., they include code that is unreachable due to data flow issues that our modeling techniques do not consider. Throughout this paper, maximal permissiveness is in the context of our program model.

III. OVERVIEW

A. Architecture

Figure 2 shows the architecture of our approach. At a high level, the approach involves two stages: *offline* analysis and control logic synthesis and *online* control. The objective of the offline analysis and synthesis stage is to generate the control logic that will ensure deadlock-free execution at run time. Namely, the feedback control logic to be synthesized

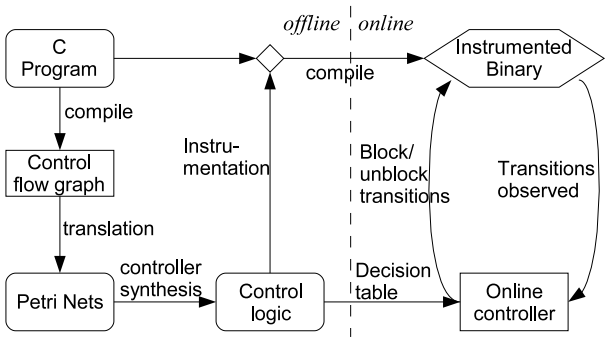

Fig. 2. Program control architecture.

will tell the controller module when to grant or postpone a lock acquisition request. This offline analysis involves several steps. The multi-threaded C program is first fed into the compiler to generate its CFG. Second, the CFG is translated into a Petri net. Third, the SBPI method is employed to generate the required control logic. As part of this step, the control logic specifies when to update system states and when to enforce control actions at run time. To properly carry out the control logic online, we must instrument the program so that it can communicate with the controller. At run time, this communication is the standard observation-action loop.

B. Challenges and Limitations

Although the idea and the structure of the approach in Figure 2 are arguably straightforward, several challenges arise when dealing with real-world programs. First, although translation of the flow information of a CFG into a Petri net is straightforward, data flow information is missing. Two problems are prominent: pointer analysis and branching analysis. Pointer analysis includes lock pointers and function pointers. Lock pointers obfuscate the actual lock instance that is acquired; furthermore locks are frequently accessed via pointers to data structures that contain locks. We use the type of the enclosing structure instead of the lock instance to approximate the model. For function pointers, we assume that every function in the scope could be referred to. Branching information is important in some situations. For example, the `trylock()` function is typically used in branches; to model it properly, we need to know which branch represents successful lock acquisition. Currently we do not model these functions.

Deadlock is a computationally complex problem. Early studies of resource deadlocks showed that determining whether a state is safe from future deadlocks is NP-complete as long as *partial resource requests* are allowed or the requests are not *linearly ordered* [8]. Here, by safety we mean there is a resource allocation scheme at the current state such that all resources will be eventually freed. In the context of manufacturing processes, which closely resemble our models, systems are classified based on the complexity of processes and the resource allocation at each stage of a process [28]. Determining the safety property in the simplest case with branchless processes and single-unit resource

requests at each stage (LIN-SU-RAS) is already an NP-complete problem. Deadlocks with mutexes are more general than LIN-SU-RAS, and therefore the problem is at least NP-hard. However, in reality, programmers often use locks in a simple way due to fear of deadlock. We have not seen any program with an individual thread holding more than five different (types of) locks at a time. In addition, the PN models translated from CFGs have special features that we can exploit to reduce the complexity of control synthesis; see Section IV-D for details.

Run-time performance is crucial; online control overhead must be minimized. There are two overheads introduced by the feedback controller. The first is the overhead of updating the controller and implementing control actions. This overhead must be minimized. The second is the overhead when a thread is blocked (i.e., delayed) by the controller on its lock acquisition call. We require that the controller postpones a lock acquisition request if and only if granting the lock might lead to deadlocks. In other words, we require the controller to be maximally permissive. Maximal permissiveness is a crucial property. One could guarantee deadlock-freeness by serializing all threads, which of course defeats the purpose of concurrent programming. Maximal permissiveness is an important reason why we employ SBPI rather than the algebraic methods described in [28], [29].

IV. IMPLEMENTATION

This section discusses implementation details of the components in our program control architecture (Figure 2). Due to space limitations, we keep the discussion at a high level.

A. Control Flow Graphs

We use the open source compiler OpenIMPACT [24] to generate CFGs from C programs. The compiler was modified such that in addition to the structure of the program, information related to lock variables and their enclosing data structures is also included. For example, the CFG records the total number of locks used, and the type of the enclosing data structure of the lock requested at each lock acquisition call. In real-world programs, wrapper structures and functions are widely used to encapsulate locks and build customized lock functions. We use annotations to denote these structures and wrapper functions so the compiler knows what needs to be included in the CFG. The number of annotations is usually small (a few lines per program).

B. Translation into Petri Nets

The CFG of each function is essentially an automaton. Therefore, we translate each function into a Petri net that is a *state machine* (i.e., each transition has a single input place and a single output place); this net represents each thread as a token flowing through it. Using the CFGs of all the functions in the program, a *linked* whole-system CFG is constructed. Locks are added to the net as *mutex places* with one initial token each. We link these places to transitions representing lock acquisition and release calls. When wrapper data structures and functions are used and annotated, the wrapper structure rather than the lock itself is represented by a mutex place.

We currently do not model thread creation and deletion because threads are created with pointers to the entry functions of the new threads. Pointer analysis is a well known difficult problem. Instead, we take a pessimistic approach by assuming every function could be executed concurrently with infinitely many threads, i.e., there is an infinite number of tokens at the entry of each function. If the program actually does not deadlock because of an insufficient number of threads, our control logic does not do any harm other than incurring a constant overhead. Additional annotations recording single or multi-threaded operation could be used.

C. Pruning

Pruning the Petri net model of the linked whole-system CFG with mutex places is a correctness-preserving performance optimization that can be made at no loss of control capabilities for deadlock avoidance. Specifically, parts of the Petri net that are irrelevant to deadlock analysis, e.g., subgraphs that contain no libpthread calls, can be deleted, i.e., abstracted out. In essence, the goal of this pruning is to shrink the model as much as possible yet guarantee that the output controller will be equivalent to the one generated by the original unpruned model.

Pruning involves two phases: function removal and function reduction. The former removes functions that do not call any lock functions directly or indirectly. We build a *function call graph* and then remove functions not connected to any lock acquisition/release function. Function reduction works inside functions that remain after the first phase and removes places and transitions that do not affect the control logic. First we identify a set of "essential" places that cannot be removed. These are places representing function calls which call lock functions directly or indirectly. Then an iterative procedure removes certain non-essential places and transitions. During each iteration, we first remove non-essential places with exactly one incoming transition and one outgoing transition, and then remove self-loop transitions and duplicate transitions, i.e., transitions linking exactly the same incoming and outgoing places. The procedure stops when no additional place or transition can be removed. We found this simple iterative procedure to be very efficient and effective, usually converging in no more than three iterations. Our pruning algorithm preserves the one-to-one mapping from each place to a basic block in the program, which facilitates the online control implementation. Further pruning or reduction techniques [21] typically violate this desired mapping and therefore are not adopted.

D. Offline Control Synthesis

The Petri nets obtained by our modeling procedure closely resemble those in the special class of Petri nets called S^3PR Nets [6]. This class was characterized in the study of manufacturing systems, and it has been extensively studied [16]. Our model actually belongs to the class called S^*PR Nets [7], which is more general than S^3PR Nets.

Nevertheless, many methods in [16] could be applied to our case, with suitable modifications. As noted in Section II, applying the SBPI method to deadlock avoidance results in an iterative procedure that may not converge [12]. Most methods reviewed in [16] are based on the SBPI iterative procedure, but with conservative placement of control places to accelerate convergence of the procedure. As a result, maximally permissiveness may be lost. In our context, even though our model is more general than S^3PR Nets, we have observed that locks in our nets are loosely coupled. There are few siphons in the original model, and siphons introduced by control synthesis are even rarer. Therefore, we have chosen to apply the standard SBPI method to guarantee maximal permissiveness. Some techniques are borrowed from existing approaches in [16] to make the procedure converge more efficiently. It is critical to identify siphons that are already controlled, and therefore do not generate new control places for them. For the remaining uncontrolled siphons, recent research [14], [15] further points out that there are dependencies among them. Controlling only a subset of these siphons, called *elementary siphons*, guarantees that the other siphons never empty. However, the controller may not be maximally permissive.

Based on these known results and on the special features of our Petri net subclass, we apply the following permissiveness-preserving heuristics for our offline control synthesis algorithm.

1) *A siphon containing fewer than two mutex places is controlled.* This is a well-known result [6].
2) *Control minimal siphons only; the minimal siphon induced by a subset of mutexes is unique.* The first part is well-known. The uniqueness of the siphon is because the Petri net model of each function itself (without any mutex place) is a state machine.
3) *Calculate siphons among circular waiting mutexes only.* A siphon induced by a subset of mutexes without circular waiting is a controlled siphon. We first traverse through the net to find lock dependencies. For every subset of locks where there is a cycle, the minimal siphon is calculated.
4) *Among equivalent siphons, control only one.* Among a set of *equivalent siphons* [15], controlling the one with minimum number of initial tokens, called the *token-poor* siphon [15], guarantees that the others never empty.
5) *Remove control places with redundant logic* Checking redundant control logic is a known difficult problem. For simplicity, we compare the control logic between newly-generated control places and existing ones. If the control logic of one control place is more permissive than that of another, i.e., its token is always available when requested, it is removed.

E. Online Control

After the offline modeling and controller synthesis steps, the resulting controller consists of a set of places with incoming and outgoing arcs to transitions in the system model. Intuitively speaking, transitions linked *to* a control place are events we want to observe. During online execution of the program, the controller state needs to be updated when such transitions fire. Transitions linked *from* a control place are events that need to be controlled. These are *controllable* events. The controller determines whether the controllable event is *enabled* or *disabled*, i.e., whether the lock request corresponding to the transition is postponed or granted immediately. Since we cannot change program execution other than delaying lock acquisition calls, not all transitions are controllable. The SBPI method can be extended to handle partial controllability [12]. The procedure involves constraint transformation and integer programming. The convergence of the procedure and the maximal permissiveness of the output controller are net dependent. For simplicity, as the Petri net without mutex places is a state machine, if a control place links *to* a transition that is not controllable, we move the link backward until a lock acquisition transition.

The program must communicate with the controller at run time to update the state of the controller and implement the control logic, that is, the enablement and disablement of the controllable events. We implemented library call interposition for online control. Library call interposition does not modify the source code. It intercepts library calls and replaces them with user-defined functions. In our case, we intercept lock acquisition and release calls, check and update the control place if necessary, and then call the actual lock acquisition and release functions. Since library interposition can intercept only library calls, it does not fully observe the program state. Extensions of the SBPI method to handle partial observability are described in [12]. Most systems we tested were observable, i.e., the control places generated had incoming links from lock acquisition/release transitions only. A more general online control method, called *program instrumentation*, can fully observe the execution of the program. It is the standard general method to obtain runtime information. Extra code invoking the controller is inserted into the program around the transitions linked with control places. Implementation of the program instrumentation method is in progress.

V. EXPERIMENTS

Random multi-threaded programs were generated with a random-walk-style algorithm. At each step, the program randomly decides either to grab a lock or to release a lock it already holds. Branches and loops are also generated similarly, such that lock acquisitions and releases are paired up in one loop or one branch. We generated 20 random programs, each with a different number of locks. The number of steps of the random walk is proportional to the number of locks. For all these random programs, we gradually added every heuristic described in Section IV-D. The results are shown in Table I. Success rates denote the portion of programs where the iterative control synthesis algorithm returns (converges) within five minutes. Once the control synthesis algorithm converged, we built an interposed library controller and ran the program. Without the controller,

Heuristics Used	Number of Locks 3		5		7		10	
1)	0%	NA	NA	NA	NA	NA	NA	NA
1)-2)	100%	1	30%	NA	NA	NA	NA	NA
1)-3)	100%	1	100%	2	75%	9	0%	NA
1)-4)	100%	<1	100%	2	100%	7	10%	NA
1)-5)	100%	<1	100%	1	100%	11	100%	63

TABLE I

SUCCESS RATE AND TIME (SEC) OF CONTROL SYNTHESIS STEP

most programs deadlock in less than a minute. With the controller, no program deadlocks within a ten-minute run. With all heuristics applied, the algorithm is able to deal with randomly generated programs with up to ten locks. We noticed that the first iteration dominates the total computation time, because it exhaustively enumerates all subsets of the set of locks. A better minimal siphon finding algorithm could be used, such as the one in [18]. We leave this for future work.

We mentioned in Section III-B that there are two overheads: controller state lookup/update and the postponing of lock acquisition calls. Experiments with randomly generated programs show that controller state lookup/update overhead is negligible, less than 1 ms for both types of implementations, instrumentation and library interposition. The overhead of postponing lock acquisition calls depends on the program, and can range widely.

VI. CONCLUSION

We have reported on our ongoing work on the application of supervisory control methods to the problem of deadlock avoidance in concurrent software. We have highlighted the challenges that arise in: (i) building suitable models of concurrent programs for the purpose of deadlock avoidance; (ii) applying existing results in supervisory control of Petri nets to synthesize controllers that avoid deadlock and are minimally restrictive; and (iii) implementing these controllers online with minimal overhead. While much work remains to be done to test the scalability of our approach, results on randomly generated programs demonstrate that the approach is technically sound and technologically feasible.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proc. SOSP*, pages 117–130, 2007.

[3] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[4] J. Edward G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[5] D. Engler and K. Ashcraft. RacerX : effective, static detection of race conditions and deadlocks. In *Proc. SOSP*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[6] J. Ezpeleta, J. M. Colom, and J. Martínez. A petri net based deadlock prevention policy for flexible manufacturing systems. *IEEE Trans. on Robotics and Automation*, 11(2):173–184, Apr. 1995.

[7] J. Ezpeleta, F. García-Vallés, and J. M. Colom. A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states. *IEEE Trans. on Robotics and Automation*, 18(4):621–625, Aug. 2002.

[8] E. M. Gold. Deadlock prediction: easy and difficult cases. *SIAM Journal on Computing*, 7(3):320–336, Aug. 1978.

[9] A. N. Habermann. Prevention of system deadlocks. *Commun. ACM*, 12(7):373–ff., 1969.

[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

[11] L. Holloway, B. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 7(2):151–190, 1997.

[12] M. V. Iordache and P. J. Antsaklis. *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach*. Birkhäuser, Boston, MA, 2006.

[13] J. Larus and R. Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[14] Z. Li and M. Zhou. Elementary siphons of Petri nets and their application to deadlock prevention in flexible manufacturing systems. *IEEE Trans. on Systems, Man, and Cybernetics—Part A*, 34(1):38–51, Jan. 2004.

[15] Z. Li and M. Zhou. Control of elementary and dependent siphons in Petri nets and their application. *IEEE Trans. on Systems, Man, and Cybernetics—Part A*, 38(1):133–148, Jan. 2008.

[16] Z. Li, M. Zhou, and N. Wu. A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. on Systems, Man, and Cybernetics—Part C*, 38(2):173–188, Mar. 2008.

[17] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proc. SOSP*, pages 103–116, 2007.

[18] J. Martinez and M. Silva. A simple and fast algorithm to obtain all invariants of a generalized Petri net. In *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*, pages 301–310, 1981.

[19] T. Minoura. Deadlock avoidance revisited. *J. ACM*, 29(4):1023–1048, 1982.

[20] J. O. Moody and P. J. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets*. Kluwer Academic Publishers, Boston, MA, 1998.

[21] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[22] G. Newton. Deadlock prevention, detection, and resolution: an annotated bibliography. *SIGOPS Oper. Syst. Rev.*, 13(2):33–44, 1979.

[23] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. SOSP*, pages 191–205, 2005.

[24] OpenIMPACT. `http://www.gelato.uiuc.edu/`.

[25] USENIX Symposium on Operating Systems Design and Implementation (OSDI). `http://www.usenix.org/events/byname/osdi.html`.

[26] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proc. ACM SIGOPS European Conf. (EuroSys)*, pages 289–302, 2007.

[27] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failure. In *Proc. SOSP*, Oct. 2005.

[28] S. A. Reveliotis. *Real-Time Management of Resource Allocation Systems: A Discrete-Event Systems Approach*. Springer, New York, NY, 2005.

[29] S. A. Reveliotis. *Algebraic Deadlock Avoidance Policies for Sequential Resource Allocation Systems*, pages 235–289. Facility Logistics: Approaches and Solutions to Next Generation Challenges. Auerbach, Dec. 2007.

[30] ACM Symposium on Operating Systems Principles (SOSP). `http://sosp.org/`.

[31] Y. Wang, T. Kelly, and S. Lafortune. Discrete control for safe execution of IT automation workflows. In *Proc. ACM SIGOPS European Conf. (EuroSys)*, pages 305–314, 2007.

[32] D. Zöbel and C. Koch. Resolution techniques and complexity results with deadlocks: a classifying and annotated bibliography. *SIGOPS Oper. Syst. Rev.*, 22(1):52–72, 1988.